Beep

The microservice way

Hugo Ponthieu

Table of Contents

Abstract

Beep is a chat application that allows users to communicate with each *other in real-time. The goal of this document is to provide an overview of the architecture of the application, including the services, protocols, and security mechanisms used to build the application.*

Currently the application is developed in a monolithic way. As the features grow, the application is harder to maintain and scale. In fact, all the elements of the application can only be scaled together. Futhermore, the application is not fault tolerant, if a service fails, the whole application will be down.

Those outage can be caused by high loads or network issues. It can also come from the development team that can introduce bugs in the application. The decoupling of responsibilities that offer microservices architecture will help to reduce the impact of a single component failure and let us to scale the application more easily.

1. Application Overview

1.1. Current features

Chatting via channels: Users can create channels and send messages to each other. They can also create private channels to communicate with their friends.

Servers: Server are the main entity of the application. They regroup users and channels. A user can be part of multiple servers. A server can have multiple channels. Inside, allowed users can communicate with each other or define roles. Web

1.2. End-user capabilities

To better design and implement the application, we will need to define the capabilities of the users. With the needs of client in mind, we will be able to define parts of the application that can be separated.

But also we will be able to define any point of dependency between the services. This will allow us to define potential interactions between the services.

We will try to define the capabilities of the user by making it impersonate different characters. There will be characters based on the authorization level of the user: - Guest user: A user that is not authenticated - User: A user that is authenticated

But also on more fine grained authorization level: - UserModerator: A user that can perform some actions on the users - ServerModerator: A user that can perform manage the server of the application

And of course there will be more fined grained authorization on resources that we will define along with the capabilities of the user.

1.2.1. Account

We will begin like if a user just discrovers the application. He will be able to access the application as a guest user.

As a guest user I want to be able to sign up using:

- My email and password
- My google account
- My Polytech account

The users that are sign up will have the abilitie to authenticate and access the application only once their account is validated.

As a user:

- I want to be able to sign in using the same methods as the sign up in order to access the application.
- that is sign in with my email and password I want to be able to link my google account to my account so that I can later authenticate with it.
- I want to change my password in order to secure my account.
- I want to be able to delete my account in order to leave the application.

Once the user is authenticated he will be able to access the application. Therefore we will focus now more on action that he can perform on his profile information.

As a user:

- I want to be able to update my profile information such as my name, last name, nickname and profile picture in order to keep it up to date.
- I want to be able to change my email in order to transfer my account to another email.
- I want to be able to change my password in order to secure my account.
- I want to be able to activate Two-Factor Authentication in order to secure my account.
- I want to be able to deactivate Two-Factor Authentication in order to secure my account.
- I want to be able to delete my account in order to leave the application.

NOTE: USE CASE DIAGRAM

1.2.2. Social

Once authenticated the user will be able to access some features without any further authorization. He will be able to access the friends system. We can first focus on the friend management for a given user.

As a user I want to:

- invite a user from their username to add a friend.
- list the invitations that I have sent or that other user sent me in order to manage them.
- accept a friend request in order to connect with the user.
- decline a friend request.
- cancel a friend request that I have sent in order to not have the user as friend.

Once I am friend with a user I want to be able to manage my frienship.

As a user I want to:

- list my friends in order to see who are my friends.
- remove a friend in order to not have him as friend anymore.

In order to regroup users, users be members of servers. There are 2 types of servers, public and private. The user can join a public server without any authorization. But he will need to be invited to join a private server. So as a user I want to:

- see all the public servers in order to join them.
- see all the servers that I am member of in order to manage them.
- leave server so that I am not related to it anymore.
- be able to answer to a server invitation so I can be a member of a server.
- browse the servers by their name and description so I can find the communities that I want to join.

NOTE: USE CASE DIAGRAM

1.2.3. Chatting

As user discover other users, he will want to interact with them. He will be able to do that through the chat system. It is composed of channel that contain messages. We will see in that part what are the abilities. As a user I want to:

- create a channel to be able to communicate with other users.
- delete a channel in order to not have it anymore.
- list the channels that I am part of in order to manage them.
- join a channel in order to communicate with the users.
- leave a channel in order to not be part of it anymore.

- add a user to a channel in order to let him communicate with the users.
- to search thrgouh the entire messages of a channel to find a message based on a keyword

With access to a channel the user will want to discuss with other users. As a user I want to:

- send a message in a channel in order to communicate with the users.
- send files in a message in order to share them with the users.
- delete a message so that I clean a channel.
- edit a message in order to correct it.
- list the messages of a channel in order to see the history of the channel.
- to pin messages in a channel to keep them visible for long time.

NOTE: USE CASE DIAGRAM

1.2.4. Severs

As cited before the user will be able to join servers. They regroup users and channels. A user that is authenticated and that as access to a particular server is called a member of the server.

By default a member will not perform any action on the server. He will need to be granted with a role to perform some actions. Role are defined at the server level and they will be aggregation of more fine-grained role.

The fine-grained roles will be:

- administrator
- server manager
- role manager
- channel manager
- channel viewer
- webhook manager
- nickname manager
- nickname changer
- message sender
- message manager
- file attacher
- member manager
- invitation manager

As invitation manager I want to:

- invite a user to a server in order to let him join the server.

- create an invitation in order to let users join the server.
- choose the expiration date of an invitation in order to manage the invitations.

As a member manager I want to:

- add a role to a member so they can perform specific actions.
- remove a role from a member to prevent them from performing certain actions.
- list the members of a server to manage them effectively.
- temporarily mute members to restrict them from sending messages.
- ban members to prevent them from joining the server.
- kick members to remove them from the server.

As a role manager I want to:

- create a role to define user permissions.
- update a role to modify user permissions.
- delete a role to remove it from the system.
- list the roles of a server to manage them.
- assign roles to members to enable them to perform specific actions.
- remove roles from members to restrict their actions.

As a nickname manager I want to:

- update the nickname of a member to change their display name.
- change my own nickname to update my display name.

As a nickname changer I want to:

- change my own nickname to update my display name.

As a channel manager I want to:

- create a channel to enable users to communicate.
- update a channel to modify its settings.
- delete a channel to remove it from the server.
- list the channels of a server to manage them.
- restrict permissions of user or role on a channel to control user actions.

As a channel viewer I want to:

- list the messages of a channel to view the conversation.
- search for messages in a channel to find specific information.
- list channel of a server to find the channel I want to see the conversation of.

As a message sender I want to:

- send a message in a channel to communicate with other users.
- update a message to correct it.

As a message manager I want to:

- delete a message to remove it from the channel.
- pin a message to keep it visible in the channel.
- perform same action as the message sender.

As a file attacher I want to:

- attach a file to a message to share it with other users.

As a server manager I want to:

- update the server settings to modify its configuration.
- delete the server to remove it from the system.
- perform the same action as the channel manager.

As an administrator I want to:

- perform all actions on the server to manage it effectively.

NOTE: USE CASE DIAGRAM

1.2.5. Administration

With the affluence of users, the application will need to be managed. The administration of resource will be done by different type of admin. This time role will be directly associated to the users.

Roles will be:

- UserModerator
- ServerModerator
- ApplicationAdministrator

2. Global Architecture

2.1. Presentation

2.2. Schema

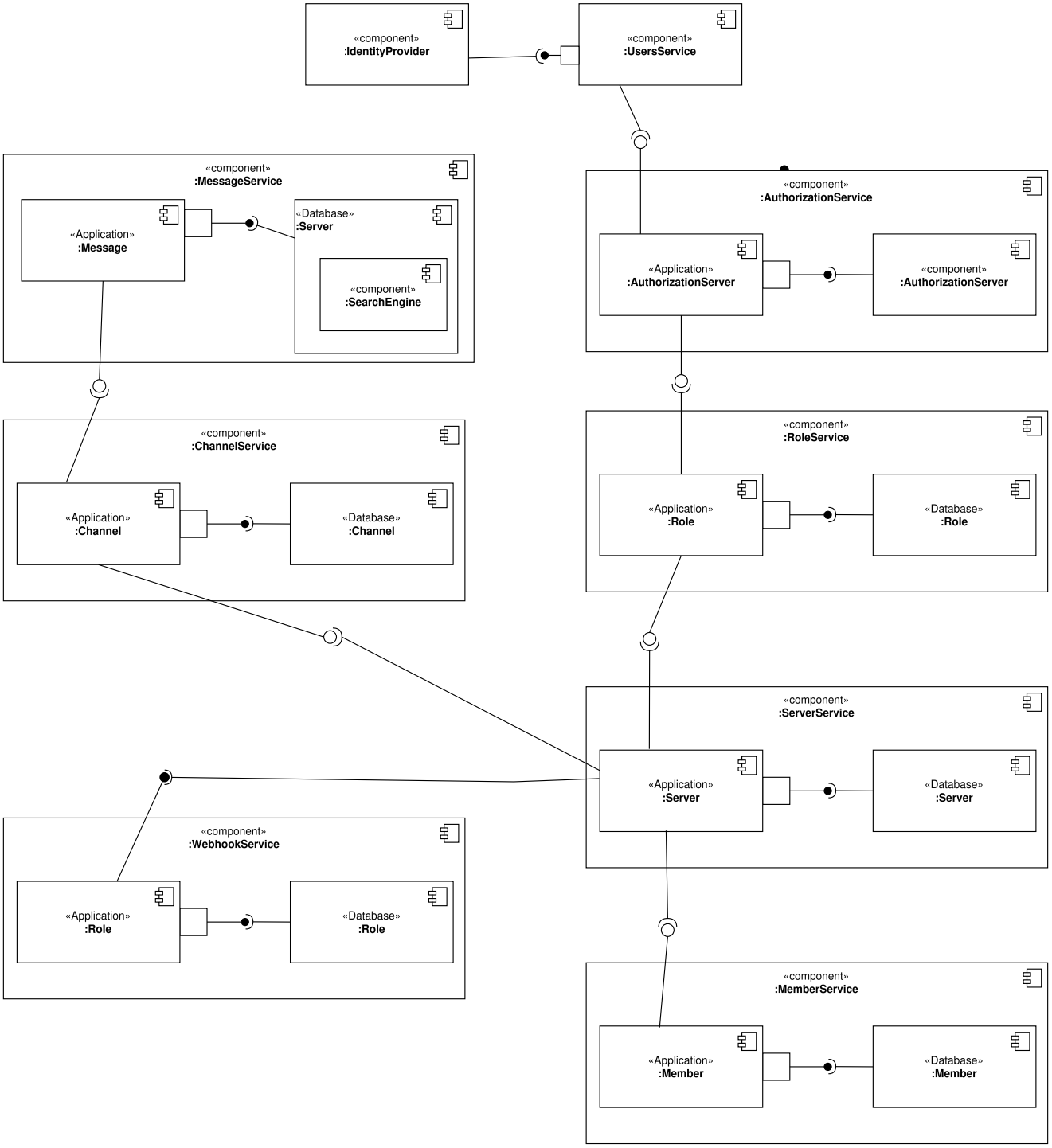


Figure 1. Overview of the application

2.3. Security

2.3.1. Authentication

Frontend integration of keycloak

Backend integration of keycloak

Introduction

Authentication is a critical aspect of any microservices architecture. In this document, we will

discuss how to integrate Keycloak for authentication to enforce authentication policies at the gateway level.

Keycloak Overview

Keycloak is an open-source identity and access management solution. It provides features such as single sign-on (SSO), user federation, and social login. Keycloak is a suitable choice for our application due to its robust authentication capabilities and ease of integration with microservices.

As the user should be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP Keycloak is suited for this task.

The service allow the user to authenticate natively from frontend implementation by exposing the login page of Keycloak. The user will be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP.

NOTE: SCREENSHOT OF THE KEYCLOAK GOOGLE NOTE: AUTHENTICATION WORKFLOW SEQUENCE DIAGRAM

It will take the responsability to:

- Register new users in the application
- To issue tokens the user through diverser methods (email, google, LDAP)
- To check the validity of a token

OAuth2 Overview

OAuth2 is an authorization framework that allows applications to securely obtain limited access to user accounts on an HTTP service by delegating authentication to a centralized identity provider, such as Keycloak.

Authorization Code Flow: This flow is suitable for applications that can securely store client secrets. It involves exchanging an authorization code for an access token.

In our architecture

For example if a user wants to access a resource on a service, the service will redirect the user to the authorization server (Keycloak) to authenticate the user. Once the user is authenticated, the server will issue an access token to the user, which can be used to access the resource. This token is short-lived and can be revoked at any time, providing an additional layer of security.

From the access token the user will be able to access the service. To enforce the check of the access token the service will use the introspection endpoint of the authorization server.

NOTE: SEQUENCE WORKFLOW FOR THE GATEWAY

We have to note that all service will have an upstream gateway that will check the access token of the user before forwarding the request to the service. This will ensure that only authenticated users can access the services.

Although the user will maybe need to be known by the service, in order to perform some actions. For example, getting the the list of its friends or direct messages. In that case the service will access directly the authorization server to get the user information.

End-user authentication

The user will be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP.

If the users try to access to the frontend wit

Deployment

NOTE: DEPLOYMENT SCHEME FOR THE KEYCLOAK IN CLUSTER

2.3.2. Authorization

Authorization is a critical aspect of any microservices architecture. In this document, we will discuss how to implement role-based access control (RBAC) within servers and global roles in our application.

Global Roles

Global roles are roles that are defined at the application level and apply to all services. They are typically used to define high-level permissions that are common across services. In our application, we will define global roles such as `admin`, `moderator`, and `user`. These roles will be used to enforce access control policies at the application level.

For this kind of roles we can use role based access control (RBAC) to define the permissions associated with each role. This will allow us to define fine-grained access control policies based on the user's role.

Then the role will be assiocated to users. The user granted with a role will be able to access the resources associated with the role.

The global wide role will be declared in the authorization server and will be used by the services to check the user's role.

For example

We can declare a "UserModerator" role that will have the abilities to:

- Restraint users from the application

We can declare "ServerModerator" role that will have the abilities to:

- Restraint users from a server
- Restraint users from a channel
- Restraint servers where users perform some actions that are not allowed

Server Roles

Server roles are roles that are defined at the server level and apply to a specific server. They are typically used to define permissions that are specific to a server.

Like in the global roles model we will have to define the fine grained access to control the access of the user to the resources of the server.

Permify

This service declare itself as Authorization as a Service. It will be used to manage the roles and the permissions of the users.

3. Implementation

3.1. Deployment

3.2. Networking

Microservices implies some networking constraints such as securing a flow of data between services, managing the load of the services, and ensuring the availability of the services.

For that task we will use Istio as a service mesh. It will allow us to manage the networking of the services in a more efficient way.

3.2.1. Service Mesh Overview

A service mesh is a dedicated infrastructure layer that provides service-to-service communication, observability, and security for microservices applications. It abstracts the network and provides a set of features that simplify the development and operation of microservices.

Service mesh provides the following benefits:

- Traffic management: control the flow of traffic between services, implement routing rules, and perform load balancing.
- Security: provides encryption, authentication, and authorization to secure communication between services.
- Observability: provides metrics, logging, and tracing to monitor the performance and health of services.

3.2.2. Istio Overview

Istio is an open-source implementation of a service mesh that provides advanced networking features for microservices applications. It integrates with Kubernetes and provides a set of tools to manage service-to-service communication, security, and observability.

One of the key advantages of using Istio is that it is actively developed and maintained by a well-

known and reputable community. This ensures that the project remains up-to-date with the latest features, security patches, and best practices. As a result, Istio is a reliable and robust choice for a microservices project that is intended to last over time.

One of the main component of Istio is the data plane that will be used to manage the traffic between the services. It will be composed of Envoy proxies that will be deployed alongside the services. All the traffic coming and leaving a pod is redirected to the Envoy proxy that will manage the traffic. This will allow to implement a lot of traffic related features such as load balancing, retries, timeouts, and circuit breaking.

Via proxies Istio is capable to log, trace and monitor natively and seamlessly the traffic between the services. This will allow to have a better observability of the application.

As describe, Istio will allow us to manage a lot of constraints outside of the services and let the services focus on their core functionalities. It is really suitable to delegate the networking constraints to a dedicated service that will manage them in a more efficient way.

3.2.3. mTLS (Mutual TLS)

Mutual Transport Layer Security (mTLS) is a security protocol that provides encryption, authentication, and integrity for communication between services. It ensures that only trusted services can communicate with each other and that the data exchanged between services is secure.

In our application, we will use mTLS to secure communication between services and prevent unauthorized access to sensitive data. By enabling mTLS, we can ensure that all communication between services is encrypted and authenticated, reducing the risk of data breaches and unauthorized access.

Istio provides built-in support for mTLS and makes it easy to enable mTLS for all services in the mesh.

NOTE: SCHEME OF THE PROXIES WITH MTLS

3.2.4. Gateway and Securing with TLS

Istio Gateway is a component that manages inbound and outbound traffic for services in the mesh. It acts as an entry point for external traffic and provides features such as load balancing, routing, and security.

In our application, we will use Istio Gateway to manage external traffic and secure communication with clients. By configuring Istio Gateway with Transport Layer Security (TLS), we can encrypt traffic between clients and services, ensuring that data is secure in transit.

Istio Gateway provides built-in support for TLS and makes it easy to configure TLS settings for services in the mesh. By enabling TLS on Istio Gateway, we can secure communication with clients and protect sensitive data from eavesdropping and tampering.

NOTE: SCHEME OF THE GATEWAY

3.2.5. Circuit Breaking

Circuit breaking is a design pattern that prevents cascading failures in distributed systems. It works by monitoring the health of services and breaking the circuit if a service becomes unresponsive or slow.

In our application, we will use circuit breaking to ensure service reliability and prevent service degradation. By implementing circuit breaking in Istio, we can detect and isolate failing services, preventing them from affecting other services in the mesh.

NOTE: SCHEME CIRCUIT BREAKING

3.3. Protocols

Poc grpc with rust

3.3.1. Overview of Protocols

Protocols are a fundamental component of microservices architecture, dictating the mechanisms by which services interact and exchange data. This section delves into the technical intricacies of various protocols, including REST, gRPC, and GraphQL, and elucidates the rationale behind selecting gRPC for our application.

HTTP/1.1, commonly used for RESTful APIs, is advantageous due to its simplicity, widespread adoption, and ease of implementation. It supports complex REST APIs and is inherently compatible with web browsers. However, it suffers from several limitations: the lack of type safety, verbosity of JSON payloads, and suboptimal performance due to the overhead of HTTP headers and the text-based JSON format. Despite these drawbacks, REST APIs can be secured using HTTPS with TLS (Transport Layer Security), ensuring encrypted communication.

REST APIs benefit from self-discoverability through OpenAPI specifications, which facilitate seamless integration and collaboration among microservices developed by disparate teams. This discoverability is crucial in a microservices ecosystem where services must interoperate efficiently.

gRPC's strong typing and contract-first approach, enforced through .proto files, ensure consistency and reliability in inter-service communication. This is particularly beneficial in large-scale microservices architectures where maintaining compatibility and preventing breaking changes are paramount.

Given the technical requirements of our application, including the need for efficient, low-latency communication and strong typing, we have chosen gRPC as the primary protocol for inter-service communication. gRPC's performance advantages, coupled with its robust type safety and support for bi-directional streaming, make it an ideal choice for our microservices architecture.

In summary, while REST have its merits, gRPC's technical superiority in terms of performance, efficiency, and type safety aligns with the demands of our application, ensuring reliable and scalable inter-service communication.

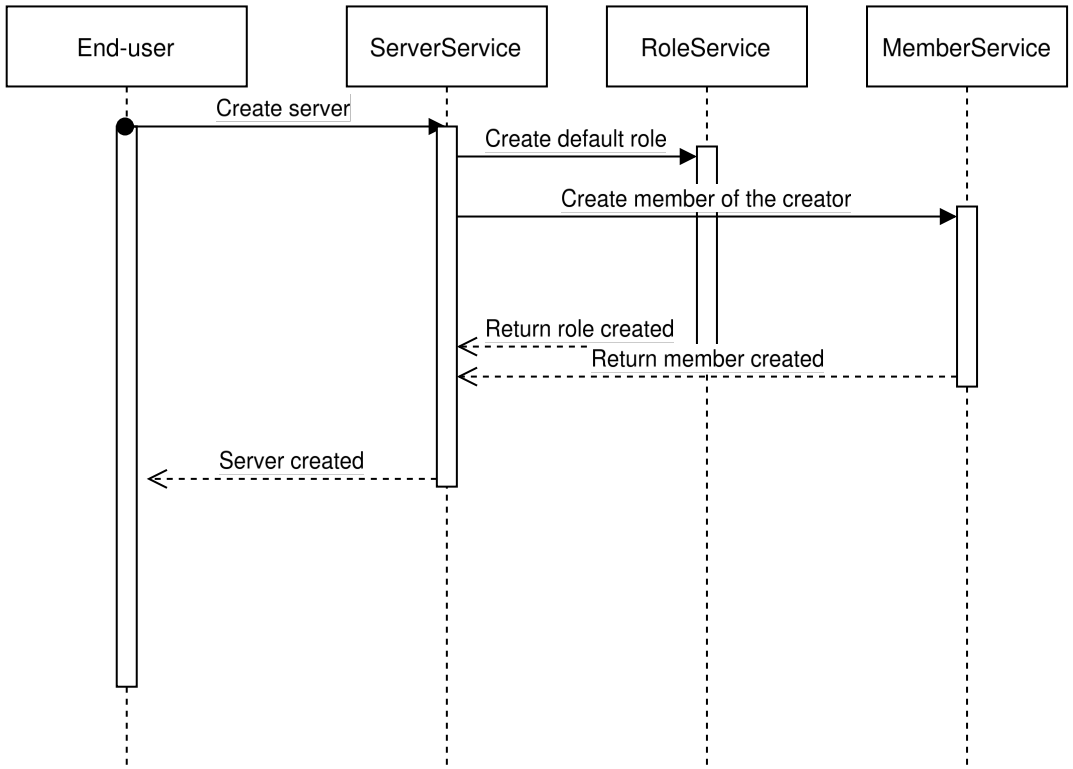
3.3.2. gRPC

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located on another computer in a network. It allows a program to execute a procedure (subroutine) in another address space (commonly on another physical machine). The calling program is suspended until the remote procedure returns, and the remote procedure executes in a different address space. RPC abstracts the communication between the client and server, making it appear as if the procedure call is local.

gRPC is a high-performance, open-source RPC framework developed by Google. It uses Protocol Buffers (protobuf) as the interface definition language (IDL) and leverages HTTP/2 for transport. gRPC offers several advantages over traditional RESTful APIs, including:

- Speed: Faster than REST due to HTTP/2, which allows multiple requests at once, compresses headers, and supports server push.
- Strong typing: Uses protobuf for data, ensuring messages are consistent and efficient.
- Real-time: Supports two-way streaming, letting clients and servers send multiple messages in real-time.
- Multi-language: Works with many programming languages, making it easy to build services in different languages.

3.3.3. Inter-service communication



3.3.4. Client communication

3.4. Services

3.4.1. Messages

Search

3.4.2. Users

3.4.3. Members

3.4.4. Roles

3.4.5. Authorization

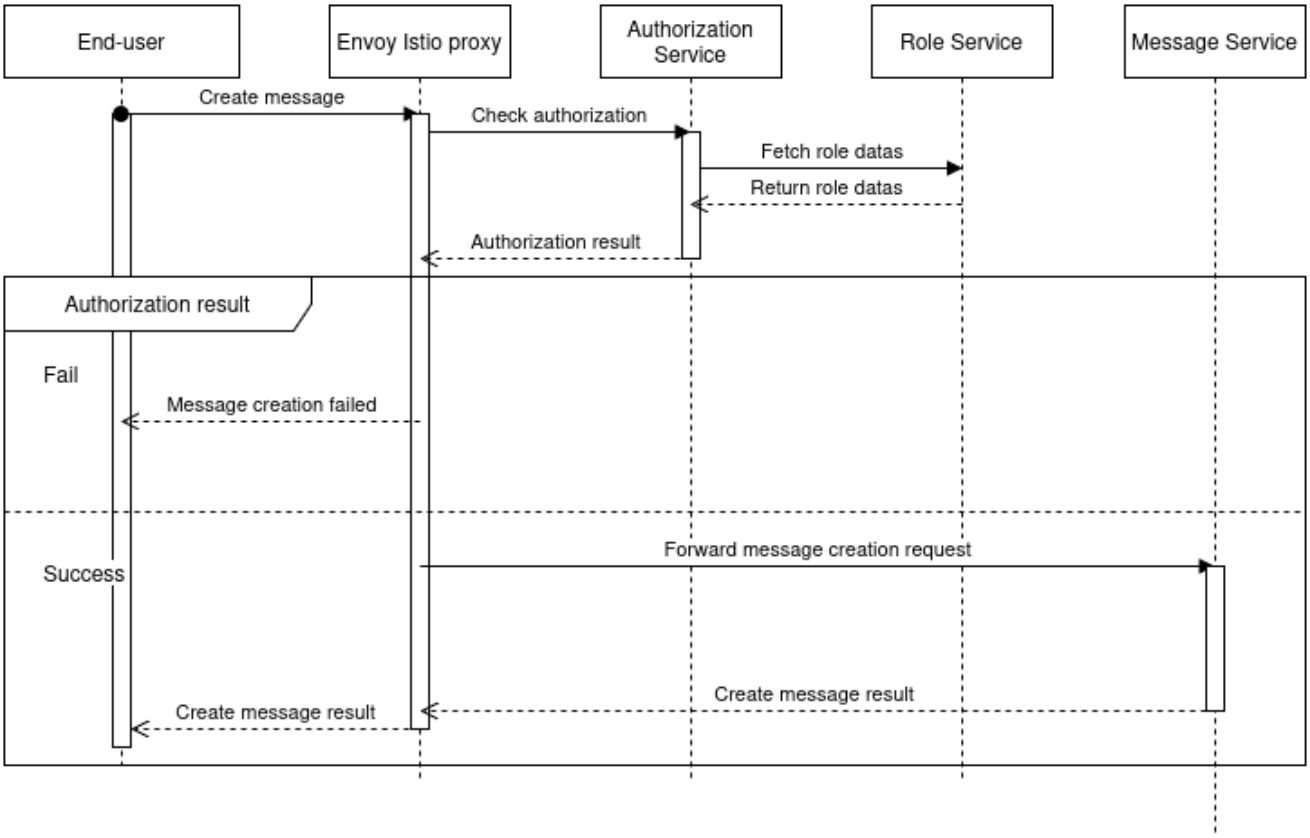


Figure 2. Create a message in a channel of a server and check the authorization

3.4.6. Servers

3.4.7. Channels

3.4.8. Messages

3.4.9. Webhooks

3.5. Backuping

3.6. Monitoring